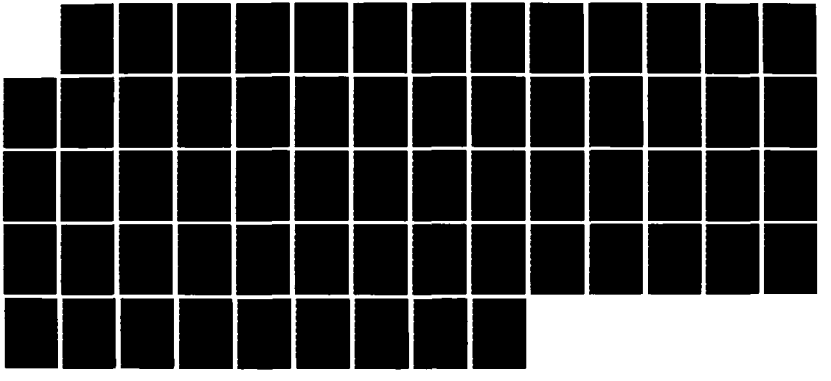
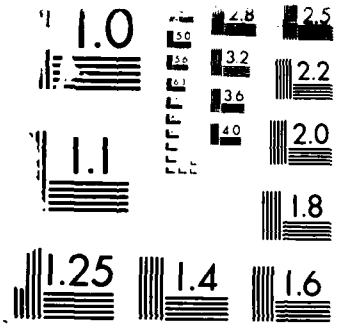
AD-A175 055 ON THE ANALYSIS OF SYNCHRONOUS COMPUTING ARRAYS(U) 1/1
STANFORD UNIV CA DEPT OF ELECTRICAL ENGINEERING
J M JOVER ET AL. 1986 AFOSR-TR-86-2171 DAAG29-83-K-0028
UNCLASSIFIED F/G 9/2 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

AD-A175 055

REPORT DOCUMENTATION PAGE

ORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
RITY CLASSIFICATION AUTHORITY NA	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
ASSIFICATION/DOWNGRADING SCHEDULE NA	
RMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S) AFOSR-TR- 86-2171
OF PERFORMING ORGANIZATION Stanford Univ sity	6b. OFFICE SYMBOL (If applicable)
	7a. NAME OF MONITORING ORGANIZATION AFOSR/NM
ESS (City, State and ZIP Code) Department of Electrical Engineering Durand 117 Stanford, CA 94305	7b. ADDRESS (City, State and ZIP Code) Bldg. 410 Bolling AFG, DC 20332-6448
OF FUNDING/SPONSORING NIZATION AFOSR	8b. OFFICE SYMBOL (If applicable) NM
	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER AFOSR 83-0228
8c. ADDRESS (City, State and ZIP Code) Bldg. 410 Bolling AFB, DC	10. SOURCE OF FUNDING NOS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.
6.1102F	2304	A6	

11. TITLE (Include Security Classification) On the Analysis of Synchronous Computing Arrays

12. PERSONAL AUTHOR(S)
J.M. Jover, T. Kailath, H. Lev-Ari and S.K. Rao.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
reprint	FROM ~~1985~~ TO ~~1986~~	1986	59

16. SUPPLEMENTARY NOTATION
1986 Workshop on VLSI and Signal Processing.

17 COSATI CODES		
FIELD	GROUP	SUB GR
xxxxxxxxxxxx		

18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
analysis of synchronous, multiple-processor systems, systolic arrays, graph-theoretical concepts, iterative algorithms.

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

see other side

DTIC FILE COPY

DEC 1 6 1986

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS ☐	UNCLASSIFIED

22a NAME OF RESPONSIBLE INDIVIDUAL	22b TELEPHONE NUMBER (Include Area Code)	22c OFFICE SYMBOL
Brian W. Woodruff	(202)767-4939	AFOSR/NM

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE

Unclassified

ABSTRACT: This paper is concerned with the analysis of synchronous, special purpose, multiple-processor systems, including, e.g., systolic arrays. There have been some results on this problem, especially by Melhem and Rheinboldt (1984). Our approach is different, combining ideas well known in linear system theory with certain graph-theoretical concepts from computer science.

A by-product of our approach to the analysis problem is a rigorous characterization of the notion of *equivalence* between iterative algorithms.

ON THE ANALYSIS OF SYNCHRONOUS COMPUTING ARRAYS†

J.M. Jover††, T. Kailath, H. Lev-Ari and S.K. Rao

Information Systems Laboratory

Stanford University

Stanford, California 94305

ABSTRACT: This paper is concerned with the analysis of synchronous, special purpose, multiple-processor systems, including, e.g., systolic arrays. There have been some results on this problem, especially by Melhem and Rheinboldt (1984). Our approach is different, combining ideas well known in linear system theory with certain graph-theoretical concepts from computer science.

A by-product of our approach to the analysis problem is a rigorous characterization of the notion of *equivalence* between iterative algorithms.

---

† This work was supported in part by the U. S. Army Research Office, under Contract DAAG29-83-K-0028, the Air Force Office of Scientific Research, Air Force Systems Command under Contract AF83-0228, the Department of the Navy (NAVELEX) under Contract N00039-84-C-0211, NASA Headquarters, Center for Aeronautics and Space Information Sciences (CASIS) under Grant NAGW-419, and the Department of the Navy, Office of Naval Research under Contract N00014-85-K-0612.

†† J. M. Jover and S. K. Rao are currently with AT&T Bell Laboratories, Holmdel, N.J. 07733.

1 Introduction

In this paper we are concerned with the *analysis problem* of determining the algorithm executed by a given synchronous, special-purpose, multiple-processor array. The problem arises because such arrays (or architectures) are often designed heuristically. Several formulations have been suggested in the computer science literature to solve a simpler problem called *verification*, in which one wants to check that a *given* algorithm is indeed implemented by the architecture.

In the analysis problem we are given the topology of the network, the function performed by each processor (including timing information), and the input data streams. We want to determine the algorithm performed by the array, and the iteration interval (*i.e.*, the time between two consecutive input samples).

Previous work in the area is due to H.T. Kung and C.E. Leiserson (1979); M. Chen and C. Mead (1982); H. Lev-Ari (1983); H.T. Kung and W.T. Lin (1983); C.J. Kuo, B. Lévy, and B. Musicus (1984); E. Tidén (1984); and, finally, R. Melhem and W.C. Rheinboldt (1984), whose work is the most general because it can determine, in some cases, the algorithm implemented by the array, instead of being limited to verifying the algorithm given. In general, these methods tend to be somewhat involved and are applicable

only to a limited class of systems. The main contribution of this paper is a new approach and a new solution to the analysis problem, based on ideas from system theory.

We view the analysis problem as part of a cycle: starting from an algorithm, we design (or synthesize) a physical circuit; then we complete the cycle (*i.e.,* solve the analysis problem) by trying to recover the original algorithm (or one that it is equivalent to it under some criterion). The first step in this cycle is to represent a given iterative algorithm, i.e., a set of relations between sequences of data, by a so-called *signal flow graph* (SFG), which shows interconnections between blocks that perform ideal mathematical operations (*i.e.*, take no time to compute). The next step in the cycle is to modify the chosen SFG to obtain a *logical circuit* (*i.e.*, a hardware implementation with physical modules that compute the same functions as the blocks in the SFG but in nonzero time and with some explicit delays). For the *analysis problem* we have to reverse the above path by modifying the logical circuit to obtain a SFG, and thereby an associated algorithm, equivalent (in some sense) to the one we started with.

Therefore to solve the analysis problem it is helpful to understand the design phase, which is our first object of attention in this section. The whole cycle will be explored in some detail using a simple example from

linear system theory; in fact this example was the one that helped us to understand the analysis problem in a context more familiar to us, since for linear systems the design and analysis problems are well understood and there are well established techniques, such as z-transforms and block-diagram-manipulations, to solve them (see, e.g., Kailath 1980).

1.1 Algorithms and SFGs

Consider the iterative expression

$$y(k) = b_1 u(k-1) - a_1 y(k-1) + b_2 u(k-2) - a_2 y(k-2) + b_3 u(k-3) - a_3 y(k-3) \tag{1}$$

which describes the relation between two sequences, $u(\bullet)$ and $y(\bullet)$, that constitute a so-called linear filter. This filter produces a sequence of output values $\{y(k)\}$, given, at each k, certain past values of $y(\bullet)$ and of an *input sequence* $u(\bullet)$. Representations of this algorithm using simple building blocks—adders, multipliers, and separators (or index-shifting blocks)—can be set up in *many ways* (see, e.g, Kailath, 1980, Ch. 2). One of these, the so-called observer form, is shown as a *signal flow graph* (SFG) in Figure 1, where we have used a convention (arising from the use of what are called z-transforms) common in system theory of labeling the separator blocks by the symbol z^{-1}.

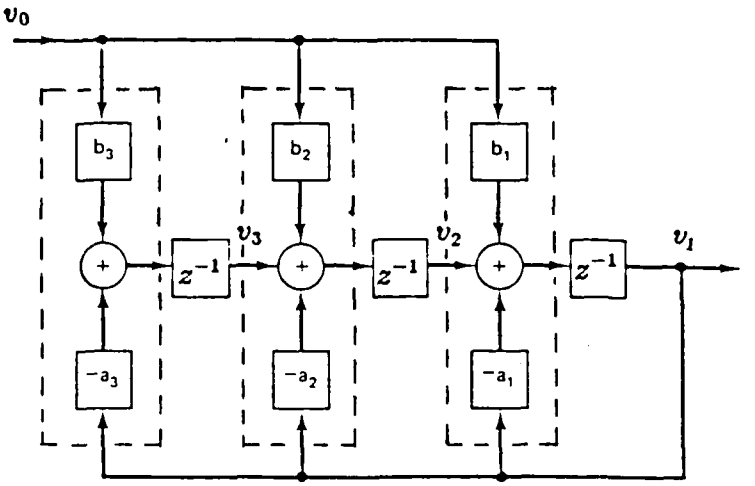


Figure 1: Observer canonical form (modified from Kailath (1980), p. 43). We define nodes $\{v_0, v_1, v_2, v_3\}$ as shown: one at the input point, and the others at the outputs of the z^{-1} blocks.

Any signal-flow-graph is a *network of connected blocks*. The interconnecting wires propagate *sequences* of data elements, which we shall call *variables*. The points at which variables appear will be called *nodes*. Thus, for instance, the variable $x_1(k)$ denotes the sequence of data elements that appears (for $k = 0, 1, \ldots$) at the output (i.e., at the node v_1) of the linear filter in Figure 1. The *processors* (=blocks) of a SFG transform one or several input variables into a single output variable. In general, this transformation need not be linear. The set of all variables and all the transformations determined by the processors constitutes the *algorithm* performed by the SFG.

Now, with the important convention that arithmetic operations are *instantaneous* (i.e., the input and output quantities have the same indices), while the separators (or z^{-1} blocks) shift the indices by unity, we can write the following (so-called "state" [1]) equations

[1] The values $\{x_1(k), x_2(k), x_3(k)\}$ describe the "state" of the system at time k, in the sense that knowing them and $\{u(l), l \geq k\}$ we can compute $\{y(l), l \geq k\}$ irrespective of the prior values of the $x_i(\bullet)$, *i.e.*, of $\{x_1(j), x_2(j), x_3(k), j < k\}$.

$$\begin{cases} x_1(k) &= b_1 u(k-1) - a_1 x_1(k-1) + x_2(k-1) \\ x_2(k) &= b_2 u(k-1) - a_2 x_1(k-1) + x_3(k-1) \\ x_3(k) &= b_3 u(k-1) - a_3 x_1(k-1) \\ y(k) &= x_1(k) \end{cases} \tag{2}$$

Notice that these state equations actually represent an *aggregated* SFG corresponding to the modules described by broken lines in Figure 1. Conversely, it is easy to see how to draw this aggregated SFG from the equations (2).

To summarize the above discussion, we can say that generally the first step in obtaining a physical implementation is to start with an input-output description and then to convert it, perhaps via the intermediate step of constructing some (aggregated) SFG representation, into an *iterative algorithm*, which is a set of equations of the form

$$\begin{aligned} x_1(k) &= f_1\{x_1(k-s_{11}), x_2(k-s_{21}), \ldots, x_n(k-s_{n1})\} \\ x_2(k) &= f_2\{x_1(k-s_{12}), x_2(k-s_{22}), \ldots, x_n(k-s_{n2})\} \\ &\vdots \\ x_n(k) &= f_n\{x_1(k-s_{1n}), x_2(k-s_{2n}), \ldots, x_n(k-s_{nn})\} \end{aligned} \tag{3}$$

where k is the (possibly multidimensional) index of iteration, and s_{ij} are known as the *index displacements*. We emphasize that this conversion pro-

cedure is highly nonunique: there are many algorithms that can implement a given input-output map. However, there is a one-to-one correspondence between equations (3) and the corresponding (aggregated) signal-flow graph.

1.2 Logical circuits

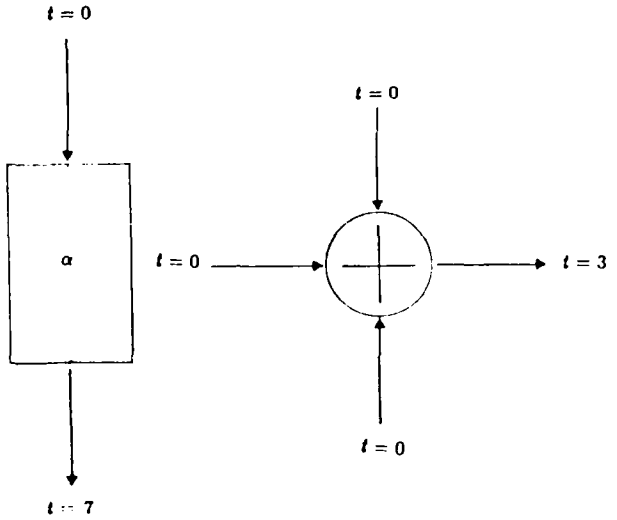
SFGs are not truly "physical" implementations of mathematical algorithms such as (2) or (3), because in any physical hardware implementation, the arithmetic operations will *not* be instantaneous. One way to accommodate these physical constraints (and to interpret the SFG as a physical system) is by taking the *iteration interval* (i.e., the physical time separation between sequence elements) to be very large, so that the arithmetic operations in each computing module will all be completed before the next iteration begins, i.e., before the next data sample is entered into the system. A more efficient procedure, likely to result in smaller iteration intervals, is to determine a "schedule" of the times at which each operation should be performed, as explained next.

We shall confine ourselves to digital implementations, in which we have an underlying clock, whose period will be taken as the basic time unit. Then the time required for additions and multiplications (or other arithmetic

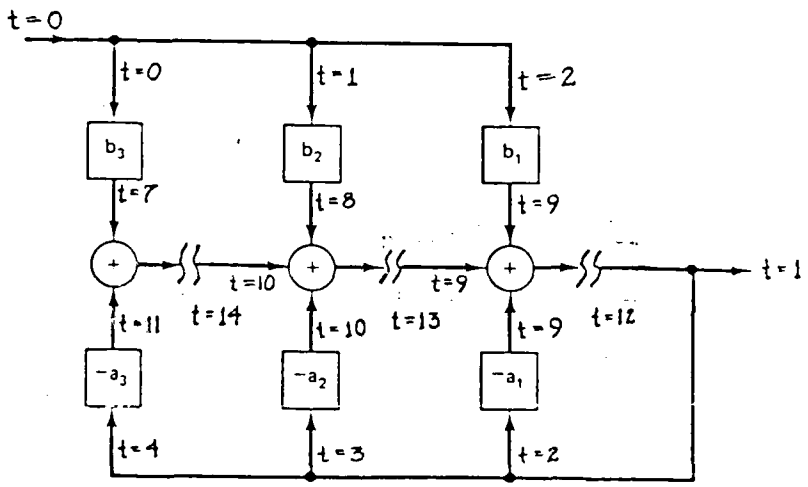
operations) will be measured as integral multiples of clock cycles. We shall not concern ourselves with the details of what happens *within* any particular clock cycle.

The main goal of the scheduling procedure is to determine an appropriate *iteration interval,* i.e., the physical time (measured in clock cycles) between two consecutive data at any point in the system (this will be the same at all points in a synchronous system), and any additional delays required, called *shimming delays*, that may have to be added to the processing and transmission delays of the system to ensure that the proper elements in the various sequences are interacting correctly.

Several algorithms for scheduling have been developed. Here we briefly describe the ideas of Jagadish *et al.*, 1985 (see also Jagadish, 1985, and Rao, 1985). Figure 2(a) shows the practical modules to implement the observer canonical form; we have assumed that multiplication (and data transfer) takes 7 clock cycles, addition (and transfer) take 3 clock cycles. We shall also assume that a pure transfer of data along an interconnecting wire takes 1 clock cycle. The z^{-1} block is a conceptual tool used to express the shifting between elements of sequences. We emphasize that such index shifts must be clearly distinguished from physical time displacements; comparison of figures 1 and 3 (to be derived) will illustrate this point.



(a)



(b)

Figure 2: Computing the schedule for the observer canonical form: (a) timing for modules, (b) computing the schedule.

Figure 2(b) shows how a schedule can be computed in our example. First, break the circuit at the z^{-1} blocks. Second, assign to the input the time 0; keep a running tab of the delays in a path to a node or to a break in the circuit. We have assigned the time 1 to the output as follows: the input data arrives at the third adder at time 9; therefore, the other inputs to this adder must be available at time 9; working backwards we find that the input to the $-a_1$ gain must be available at time 2, thus the output of the global system must be ready at time 1. We can now compute the times at the outputs of gains $-a_2$ and $-a_3$, which turn out to be 10 and 11, respectively.[2]

Third, compute the difference in values at the two sides of the breaks in the circuit: we have $14 - 10 = 4$, $13 - 9 = 4$, and $12 - 1 = 11$. The largest of these differences will define an appropriate iteration interval, δ; in our example $\delta = 11$. It must then be clear that we must add some so-called shimming delays of values 4 and 2 to the outputs of gains b_3 and b_2

[2] In graph theoretical terms what we have done so far is to solve a single source, longest path problem for the SFG. The longest path to a node clearly determines the time at which the node can be 'scheduled.' To be precise, certain preliminary separator transformations have to be carried out in order to make the single-source largest paths problem meaningful; in particular, we have to ensure that when the separators are broken we have an acyclic graph.

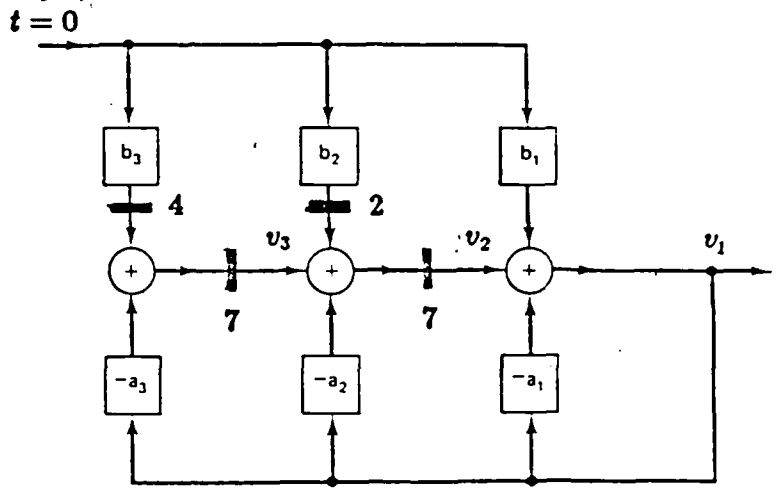


Figure 3: Logical circuit for the observer canonical form.

to ensure that all the inputs to each adder are present at the same time (11 for the far left adder, and 10 for the adder in the middle). Figure 3 shows these shimming delays as thin black blocks. Now close the gap at the breaks as follows: when the time difference at the gap is equal to the iteration interval, close the gap without any extra delay; otherwise, close the gap with a shimming delay equal to the iteration interval minus the time difference in that gap.

A physical implementation of the observer canonical form is shown in Figure 3, in which the blocks represent hardware components with a computational delay as shown in Figure 2(a), and with additional (shimming)

delays whose value (in number of clock cycles) is indicated next to them. Digital designers usually call such a figure a logical (circuit) diagram.[3] It should be noted that the scheduling procedure is highly nonunique (see Jagadish, 1985, and Rao, 1985) and therefore several different logical circuit diagrams can be associated with a given SFG. We shall elaborate on this nonuniqueness in the following subsection.

1.3 Logical graphs and algorithm graphs

For many purposes, especially timing analysis, it is convenient to redraw the logical circuit diagram as what we shall call a logical graph G, see Figure 4. To draw G, we put down a vertex of the graph for each node of the logical circuit, and connect the vertices with edges that represent the directed paths between the nodes of the logical circuit. With each edge we associate a weight corresponding to the total computational and propagation delay for that path. For example, from v_2 to v_1 we have an edge with weight 3, a self loop from v_1 to itself with weight $1+7+3=11$, a loop from v_1 to v_2 with weight $1+1+7+3+7=19$, and so on. With each vertex v_i, we associate a sequence $\{x_i\}$. For example, for the observer

[3]The adjective 'logical' arises from the fact that the hardware is based on so-called 'logical' components obeying the rules of Boolean logic (algebra).

form we can write the function performed at each vertex as follows

$$\begin{cases} x_1() &= b_1 u() - a_1 x_1() + x_2() \\ x_2() &= b_2 u() - a_2 x_1() + x_1() \\ x_3() &= b_3 u() - a_3 x_1() \\ y() &= x_1(). \end{cases} \tag{4}$$

Note that we cannot write directly from the logical graph G the actual index dependences as we did in eq. (2) from the SFG. Finding these index dependencies (i.e., the index displacements s_{ij} in (3)) and the iteration interval comprises the analysis problem.

Algorithm graphs

To facilitate the reconstruction of an iterative algorithm from a logical graph G of some physical implementation, we have to obtain a representation of iterative algorithms that is similar in form to the logical graph. For this purpose we introduce a so called *algorithm graph* G^*: this, like G, has a vertex for each variable in the algorithm, and its edges represent the index dependencies, i.e., the weight of the directed edge connecting v_j (the vertex representing $x_j()$) to v_i equals the index displacement s_{ij}. Thus, *the algorithm graph is a precise image of the iterative algorithm (3)*, i.e., there exists a one-to-one correspondence between iterative algorithms and their graphs. The difference between the algorithm graph G^* and the SFG is

that in G^*, we focus only on the index dependencies ignoring the details of the actual functional computations. It is the determination of a correct set of index dependencies from the timing information in the logical graph that is the key to the solution of the analysis problem.

Relating G and G^*

The logical graph G and the algorithm graph G^* are closely related. Obviously, G and G^* have the same topology; the only difference is that in G^* the edge weights represent the number of separators in the path while in G they represent physical delay. Moreover, there is a simple relation between the index displacement s_{ij} (=number of separators) associated with an edge in G^* and the physical delay d_{ij} associated with an edge in G. To make this relation explicit we need to analyze the way synchronous systems work.

In synchronous systems the time between two consecutive elements in any sequence is constant (and equal to the iteration interval, δ). If in addition we assume that such systems are time-invariant, as we do in the logical graphs, all the computations (at the vertices) involve data arriving at some multiple of the iteration interval. Consider, for example, the graph G for the observer form (Figure 4). Apply the first element, $u(1)$, of an input sequence $u(\bullet)$ at time $\lambda_0 = 0$; by definition, the rest of the elements will

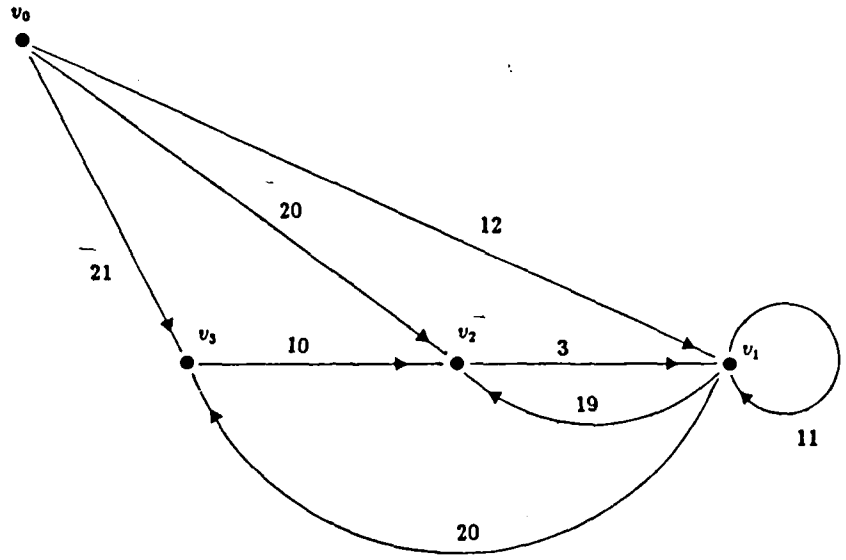


Figure 4: Logical graph G for the aggregated observer canonical form.

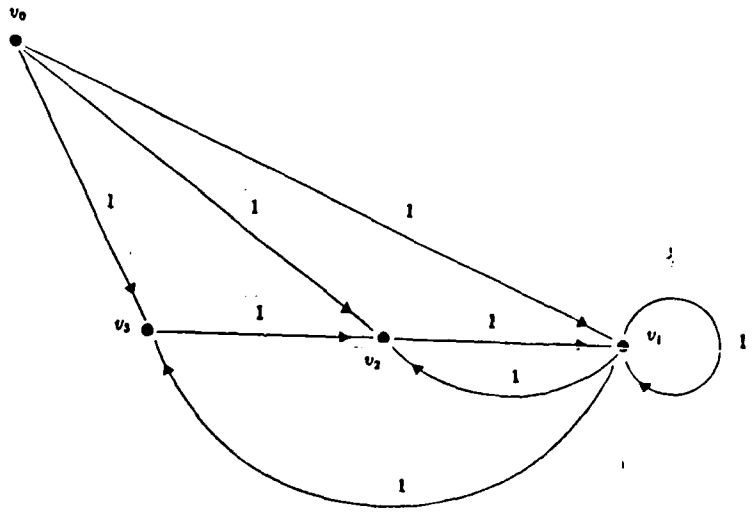


Figure 5: Algorithm graph G^* for the aggregated observer canonical form.

be generated every δ clock cycles. Let us denote by λ_i the time instant at which the vertex v_i generates the output $x_i(1)$.

Recall that d_{ij} (resp. s_{ij}) is the physical delay in the logical graph G (resp. the index displacement in the algorithm graph G^*) along a directed path connecting the vertex v_i to the vertex v_j. Since v_i generates $x_i(1)$ at time λ_i, $x_i(1)$ arrives at vertex v_j at time $\lambda_i + d_{ij}$. However, we cannot equate λ_j to $\lambda_i + \delta_{ij}$, because λ_j will depend upon the number of separators in the path from v_i to v_j, which in turn will fix the actual iteration in which the input $x_i(1)$ is operated on at vertex v_j. In our case, the path from v_i to v_j has s_{ij} separators and therefore, the vertex v_j will associate the input $x_i(1)$ with the $(1 + s_{ij})$th iteration rather than with the first iteration. Consequently, v_j will generate $x_j(1)$ at time $\lambda_j \equiv \lambda_i + d_{ij} - s_{ij}\delta$ where δ denotes the iteration interval. It follows that, for *every path* (v_i, v_j),

$$d_{ij} = (\lambda_j - \lambda_i) + s_{ij}\delta \tag{5}$$

This is the basic equation; all logical graphs G that implement a given algorithm graph G^* satisfy this condition.

After this discussion, the analysis problem can now be restated as follows: given a logic graph G, find an algorithm graph G^* and a set of λ_i[4]

[4]Only the λ_i associated with the input vertices are known apriori.

that satisfy (5).

1.4 An analysis procedure

We now present an analysis procedure that constructs nonnegative s_{ij} and a set of λ_i for a given logical graph G. First we form a *rooted tree* consisting of the shortest paths from the input vertex v_0 to the remaining vertices of G. Next we set $s_{ij} = 0$ for the edges contained in the rooted tree. This determines λ_i for all vertices and we can use equation (5) to compute the index displacements s_{ij} for the edges that are not in the rooted tree.

As an example, in the observer form of Figure 4 the first input will reach vertex v_2 from v_0 at time $\lambda_2 = 20$ (minimum-delay path). Along the path that goes through v_3, however, this first input will not reach v_2 until time 31 (i.e., $21 + 10$). But at time 31, we shall already have at vertex v_2 the *second* datum, $x_2(2)$, that traveled along the shortest path. In other words, $x_2(2)$ is a function of $u(2)$ and $x_3(1)$. Hence, we have to put one z^{-1} block in the longer path. Figure 6 shows the graph G^* with the number of separators in each edge; from this graph we can easily write the following state equations

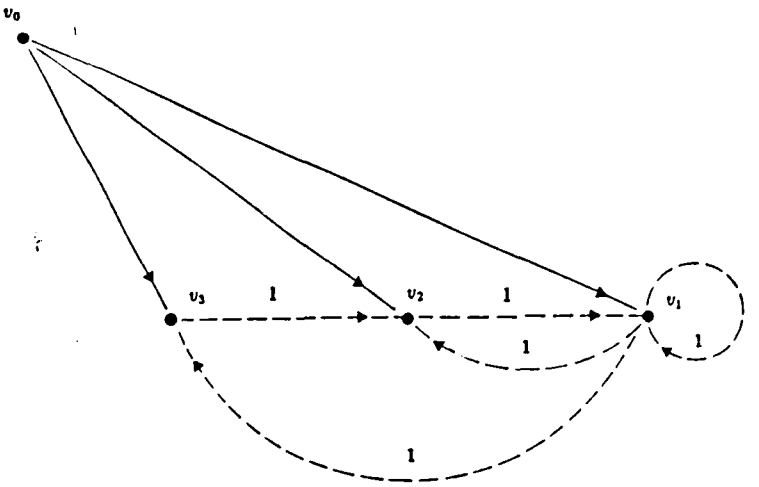


Figure 6: Another graph G^* for observer canonical form.

$$\begin{cases} x_1(k) &= b_1 u(k) - a_1 x_1(k-1) + x_2(k-1) \\ x_2(k) &= b_2 u(k) - a_2 x_1(k-1) + x_1(k-1) \\ x_3(k) &= b_3 u(k) - a_3 x_1(k-1). \end{cases} \tag{6}$$

When the logical graph has several input vertices we shall need to *extend* the logical graph by introducing an auxiliary root, as described in Section 3.

We now observe that the graphs G^* in Figures 6 and 5 are different, and consequently that the resulting algorithms (eqs. (2) and (6)) are *apparently* different. However, the difference in the algorithms in eqs. (2) and (6) amounts only to shifts in the indices (we can change the indices for the

sequence $u(\bullet)$ in eq. (6) from k to $k-1$ and obtain eq. (2)) and therefore both algorithms are *equivalent* in the sense that they determine the same set of recursions.

This observation holds for every algorithm graph G^* and for arbitrary shifts in the indices of data sequences. In other words, if we redefine $\tilde{x}_i(k) := x_i(k-c_i)$ and replace $x_i(\cdot)$ by $\tilde{x}_i(\cdot)$ in the algorithm graph, the computations remain unchanged, even though the index displacements become modified, i.e., $s_{ij} \rightarrow s_{ij}+c_i-c_j$. We shall say that two algorithm graphs are *equivalent* if they are isomorphic (in the graph theoretic sense) and if their index displacement can be related by a set of shifts c_i, as described above.

The above discussion has been informal. We shall present and prove the general procedure to obtain the algorithm graph G^* in Section 3. It is based on determining the shortest-path tree (minimum delay), which is consistent with the remark in footnote 2 that the inverse problem (viz. the scheduling problem) involves solving a single-source longest-paths problem). The shortest path procedure guarantees that the number of separators we obtain in the links of the tree is integral and positive.

2 Formulation of the Analysis Problem

In this section we discuss in more detail some of the key concepts presented in Section 1. The first subsection is dedicated to presenting a model for synchronous, special-purpose systems. In the second subsection we refine the concept of iteration interval. In the third subsection, we formalize the concept of equivalence for a pair of SFGs that have the same topology and functionality, but that differ in the weights on their edges (different distribution of separators).

2.1 Logical Graphs

As it is well known in circuit theory (see, *e.g.,* Rohrer, 1970), physical circuits can be modeled by a directed graph. For instance, Leiserson and Saxe (1981) modeled circuits as a graph in which the vertices are the computing elements and the edges the interconnections between these elements. The edges have weights associated with them, representing the number of shift registers present in the interconnection. Moreover, the vertices have also weights associated with them, representing the computational delay.

To model a synchronous circuit, Leiserson, Rose, and Saxe (1983) impose non-negativity constraints on the weights associated with the edges and vertices. In addition they require that, in any directed cycle in the

graph, the sum of the edge weights must be positive. The need for such a constraint for digital circuits was shown by Crochiere and Oppenheim (1975); see also previous work by Karp and Miller (1966).

Below we model a synchronous, special-purpose system as a finite, directed graph. For our needs, we lump the propagation and computation time in a weight associated with the edges, which represent the interconnections between variables or sequences (which are the vertices of the graph). We impose positivity constraints on the edge weights as in the previous models. To describe synchronism, we associate a data sequence to each vertex and impose the constraint that the time between two consecutive data in all sequences must be a constant, which we call iteration interval.

Definition 1 *A synchronous, special-purpose system is a finite, directed graph* $G = (V, E, x, d, \delta)$, *in which*

- V *is the set of vertices of the graph. They represent the sequences (or variables) in the circuit.*
- *There is at least one vertex in* V *(called a source) with no incoming edges.*
- x *is the set of sequences* $\{x_i\}$ *associated with the vertices (one per vertex)*
- E *is the set of directed edges, which represent the dependence among the variables so that for vertex* v_i, $x_i(k)$ *is a function (invariant in* k*) of the sequences corresponding to the vertices that have edges incoming into* v_i.

- *d is the set of positive weights (delays) associated with the edges (one per edge), which represent the processing and propagation delay corresponding to that edge.*
- *δ is a positive constant (called iteration interval) that represents the time between two consecutive elements in the sequence associated with any vertex v_i.*

Remarks:

1. The inputs of the system are the sequences associated with the source vertices. The outputs are the sequences corresponding to sinks in the graph (*i.e.,* vertices with no outgoing edges); in addition, we can designate the sequence associated with any vertex as an output of the system. When we want to show clearly that we are referring to an input or output, we shall denote the corresponding sequences with the letters u and y, respectively. A subindex indicates the vertex we are referencing; an integer in parenthesis designates a particular element in the sequence. For example, $x_3(7)$ is the seventh element in the sequence generated at vertex v_3.

2. In our model, the delays, d_{ij}, and the iteration interval, δ, are integer multiples of the clock period.

3. We use the convention of assigning the same index $k = 1$ to the first element generated at each of the vertices v_i.

4. Till we state the contrary, we assume that every vertex in G can be reached by a directed path from at least one input.

2.2 The Iteration Intervals

There are several possible notions of iteration interval for a synchronous system G. The first one, δ, is the time between two data in the input streams. As we shall see below, there might be several possible δ's that work (are valid) in a system; we establish the conditions for validity in Theorem 1 below.

Under certain conditions (see Jover and Kailath, 1984), a system can perform an algorithm on different sets of data by interleaving data from the different sets to form the input sequences for the system. Naturally, in that case the throughput increases to a value given by the number of sets interleaved times the throughput without interleaving.

The second type of iteration interval will be denoted δ_1 and refers to the time between data in a sequence such that the system is operating on one sequence, not on interleaved sequences.

There exists a third iteration interval, called the intrinsic iteration interval, δ_{intr}, which corresponds to the minimum time interval before we can introduce another datum to the same elementary module. Note that δ_{intr}

can be the clock period or a multiple of it depending on the implementation. In a practical system we let δ_{int} be the *largest* of the intrinsic iteration intervals of any elementary module. Obviously, a practical system must satisfy $\delta \geq \delta_{int}$ and $\delta_{int} \geq$ *clock period*, since we cannot introduce data faster than the clock period of a system. Finally, note that δ_{int} is technology and implementation dependent, while δ depends only on the topology and delays in the graph G.

Below we present some theorems concerning the iteration intervals. We say that a graph G has a *valid* δ when a *given* value δ for the iteration interval satisfies the definition of a synchronous system (and, of course, $\delta \geq \delta_{int}$). The following theorem provides a method to test if a graph G has a valid δ, *i.e.,* an iteration interval that makes the appropriate elements of the sequences to meet.

Theorem 1 *1. A given value δ is a valid iteration interval for a synchronous, special-purpose system $G = (V, E, x, d, \delta)$ if, and only if, the delays on all the paths between any two vertices are congruent*[5] *mod δ and the delays for self-loops are a multiple of δ.*

[5]Two numbers Ω_1 and Ω_2 are defined to be congruent modulo δ if their difference is a multiple of δ.

2. *δ_1/δ represents the number of sequences interleaved, where δ_1 is the largest valid iteration interval.*

Proof:

1. Consider one vertex v_i, and assume that there are, say, r paths between this vertex and another vertex v_j. Call the delays on the paths $d_1, \ldots d_r$. If members of the sequence x_i are sent every δ units of time starting at time $t = 0$, then the member $x_i(k)$ will reach v_j at the following times $t_1 = k\delta + d_1$, $t_2 = k\delta + d_2$ and so forth until $t_r = k\delta + d_r$. Since the delays are congruent modulo δ, their difference is a multiple of δ. Thus the differences between any times $\{t_1, \ldots t_r\}$ is also a multiple of δ, which proves that the iteration interval for the sequence in v_j is also δ. The same argument applies for self-loops. Since our discussion was for any v_i and v_j, our results are general for any two vertices. Finally, note that if we were given the fact that the δ is valid (*i.e.,* constant for the sequence at any vertex), then the differences in $\{t_1, \ldots t_r\}$ are multiples of δ, which implies that the differences in the delays $\{d_1, \ldots d_r\}$ are also a multiple of δ and therefore congruent modulo δ. This completes the proof of (1).

2. Assume a system with only one pair of vertices connected by two parallel edges in the same direction. For such a system, it is clear that δ_1

is given by the difference in the delays of both paths. Choosing an iteration interval that is a submultiple of δ_1 will result in independent sequences being processed (*i.e.,* interleaving). The number of sequences being δ_1/δ as it is trivial to check.

For a general system, if there is a pair of vertices for which the iteration interval is not the largest valid, then there will be interleaving for that pair and, therefore, this value of iteration interval will not be δ_1. Thus, δ_1 has to be the largest valid iteration interval, which proves the first property. Similarly, for the number of sequences interleaved in a pair of vertices will be given by the quotient between δ_1 and the value that makes this pair congruent. Obviously, for the general system, the largest of this quotients is the number of sequences interleaved in the whole system.

□

2.3 Algorithm Graphs

Definition 2 *An algorithm graph is a finite directed graph* $G^* = (V, E, x, s)$ *in which* V, E, x *are the same as in Definition 1, and* s *is the set of positive weights (separators) associated with the edges (one per edge), which*

represent the index displacement corresponding to that edge.

Theorem 2 *A synchronous system $G = (V, E, x, d, \delta)$ is an implementation of an algorithm graph $G^* = (V, E, x, s)$ if, and only if, there exists a set of positive constants $\{\lambda_i\}$ (one per vertex) such that for every v_i, $v_j \in V$,*

$$\lambda_j = \lambda_i + d_{ij} - s_{ij}\delta$$

Proof: See the argument used to establish eq. (5) □

The constants $\{\lambda_i\}$ determine a *schedule* for G in the sense that $x_i(k)$ is generated at the time-instant $\lambda_i + (k-1)\delta$.

Definition 3 *Two algorithm graphs $G_a^* = (V, E, x, s_a)$ and $G_b^* = (V, E, x, s_b)$ (and their corresponding algorithms) are said to be equivalent if there exists a set of constants $\{c_i\}$ (one per vertex) such that*

$$(s_b)_{i,j} = (s_a)_{i,j} + c_j - c_i \tag{7}$$

Figure 7 depicts two equivalent graphs G^* for the aggregated observer form introduced in Section 1. The main point is that two algorithms can be equivalent even if the indices do not seem to match. The important fact is to have the indices *for the same sequence* related in the same way. Indices between different sequences can always be matched by selecting an appropriate origin for the different indexing variables.

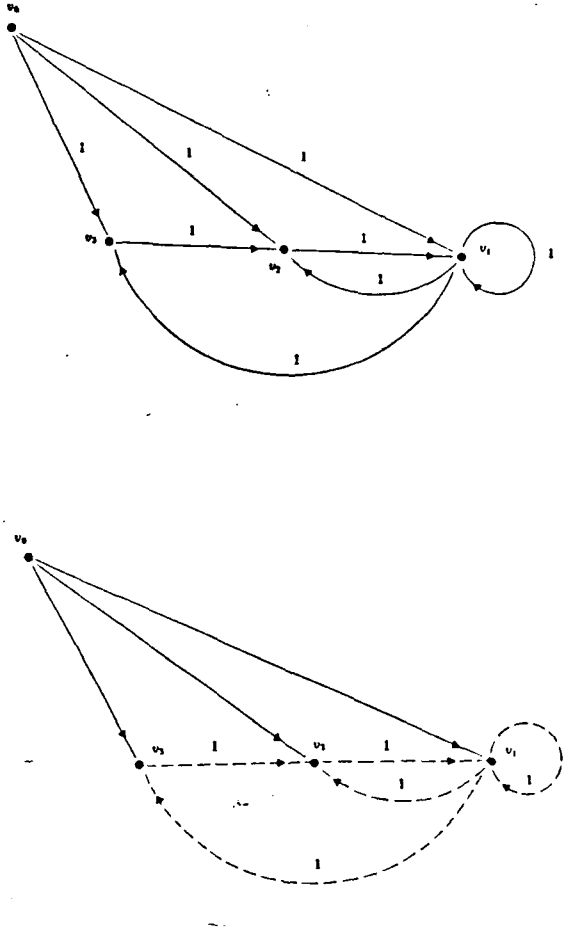


Figure 7: Two equivalent graphs G^* for the observer form.

3 An Analysis Procedure

In this section we describe our analysis procedure and provide a rigorous proof of its validity.

Procedure 1 *Given a logical graph $G = (V, E, x, d, \delta)$ we want to determine an algorithm graph $G^* = (V, E, x, s)$, such that G is an implementation of G^*. For each input sequence, $\{u_i\}$, we are given the time λ_i at which the first element of the sequence is entered. The procedure consists of the following steps:*

1. *Extend the graph G by adding a vertex v_0 with an edge from v_0 to each of the source vertices v_i, and with weight λ_i. Call this graph G_{ext}.*
2. *Form a spanning tree with the minimum-delay path from v_0 to every vertex in G_{ext}. For every vertex v_j let $\lambda_j = \lambda_i + d_{ij}$ where d_{ij} is the delay on the edge that connects to v_j its closest vertex, v_i.*
3. *For each link e_{ij} in the tree formed above compute*

$$d^*_{ij} = \lambda_i - \lambda_j + d_{ij}.$$

For each self-loop with weight d_{ii} assign

$$d^*_{ii} = d_{ii}.$$

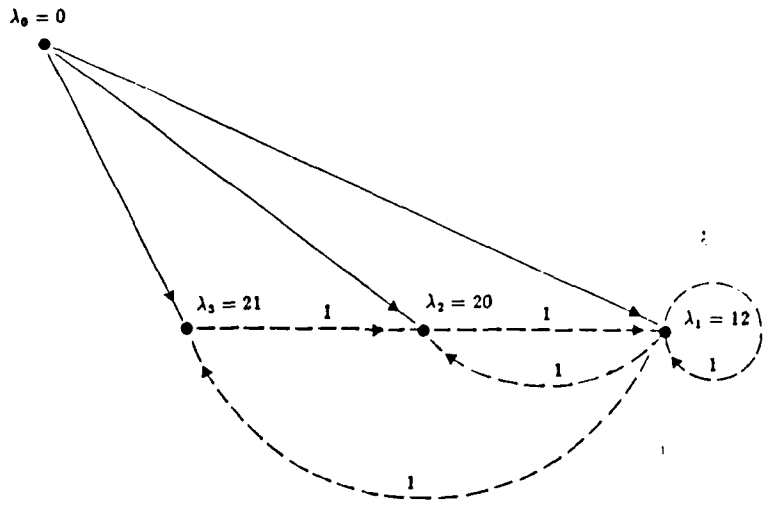


Figure 8: Shortest-path tree for observer form.

4. *Compute* $\delta_1 = gcd\{d^*_{ij} \quad \forall i, j\}$.

5. *The number of sequences interleaved is given by* δ_1/δ. *If* $(\delta_1/\delta) \notin Z^+$ *then the* δ *given is not valid for the graph* G.

6. *Associate a weight* $s_{ij} \geq 0$ *to each link* e_{ij} *in the tree formed in step 2*

$$s_{ij} = d^*_{ij}/\delta_1.$$

Associate zero weight, $s_{ij} = 0$, *to the edges in the tree.*

As a simple example recall the observer form of Section 1. Notice that an extension (step 1) is not required, because $\lambda_0 = 0$; the corresponding shortest-path tree and the resulting index displacement are described in Figure 8.

Remarks:

1. In addition to the logical graph G, we are given the time instants λ_i at which the first element from each of the input sequences $\{u_i\}$ is entered. If these time instants were not given, the procedure would still work, but information on the relative shift between input sequences could not be recovered. (As we noted in Section 2.3 the SFG recovered will still be equivalent to the one used in the design of the logical graph G.)

2. If the iteration interval δ is not given, our procedure still will work and will compute a valid δ for that system, under the assumption that there is no interleaving of input sequences. In this case there is no need to perform step 5 of the procedure.

3. There are several algorithms for implementing step 2 of the procedure (shortest-path tree), such as those in Dantzig (1975). For a comprehensive presentation of these algorithms see, *e.g.*, Even (1979).

We now turn to establish the validity of our analysis procedure.

Proof of the Analysis Procedure

We establish Procedure 1 in three stages. First, to prove steps 1 through 5 of the procedure we prove that the procedure determines whether the δ

given is valid and that it computes δ_1 correctly. Second, we prove step 6 of the procedure, *i.e.*, that the number of states assigned to a link will be nonnegative. Third, we prove that the procedure recovers an algorithm that is equivalent to the original algorithm used in generating the synchronous system.

Computing a valid δ

In this section we prove that steps 1–5 of Procedure 1 are correct. We will prove first that a valid δ is a divisor of the delays $\{d^*_{ij} \quad \forall i,j\}$. Every time we add a link e_{ij} from v_i to v_j, we are creating two paths to reach v_j (see Figure 9); one of the paths has delay $\lambda_i + d_{ij}$ and the other λ_j. Applying Theorem 1 we see that these delays must be congruent modulo δ; therefore, their difference (which we call d^*_{ij}) must be a multiple of δ.

To prove that $\delta_1 = gcd\{d^*_{ij} \quad \forall i,j\}$ just consider that any valid δ has to be a divisor of $\{d^*_{ij}\}$. By Theorem 1, δ_1 must be the largest valid δ, which completes our proof. □

Determining the states

In this section we prove that step 6 of Procedure 1 generates index displacements that are positive and integer.

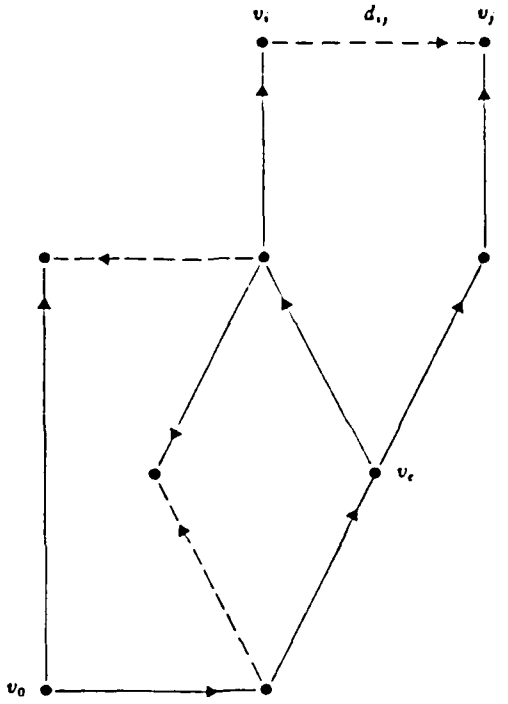


Figure 9: Graph with a tree and the corresponding links.

To prove that $s_{ij} \geq 0$ note that s_{ij} will be nonnegative if and only if $d^*_{ij} \geq 0$. Since we have formed a shortest-path tree from v_0, it means (again see Figure 9) that

$$\lambda_i + d_{ij} \geq \lambda_j$$

and, therefore,

$$d^*_{ij} = \lambda_i - \lambda_j + d_{ij} \geq 0.$$

Thus $s_{ij} \geq 0$. Finally, since δ_1 is a divisor of every d^*_{ij}, it is obvious that s_{ij}, as computed in Procedure 1, will also be integer. □

Recovering the equations

In this section we prove that Procedure 1 recovers an algorithm G^* that is equivalent to the equations used in the design of the logical graph G.

The algorithm G^* recovered by our procedure satisfies the equation $\lambda_j = \lambda_i + d_{ij} - s_{ij}\delta$, while the original algorithm $\tilde{G}^*$, used to design the logical circuit G, satisfies the equation $\tilde{\lambda}_j = \tilde{\lambda}_i + d_{ij} - \tilde{s}_{ij}\delta$. Assuming all graphs in consideration have been extended, we can assume for the root vertex v_0 that $\lambda_0 = 0 = \tilde{\lambda}_0$. Consequently,

$$\tilde{\lambda}_i - \lambda_i = (s_{i0} - \tilde{s}_{i0})\delta$$

so that we can replace $\tilde{\lambda}_i$ by $\lambda_i + c_i\delta$, where $c_i := s_{i0} - \tilde{s}_{i0}$, and we conclude

that

$$\tilde{s}_{ij}\delta = \tilde{\lambda}_i - \tilde{\lambda}_j + d_{ij} = \lambda_i - \lambda_j + d_{ij} + (c_i - c_j)\delta = (s_{ij} + c_i - c_j)\delta$$

which establishes the equivalence between G^* and $\tilde{G}^*$ □

This completes the proof that our procedure is indeed correct in each of the steps and that it recovers an algorithm equivalent to the one used in the design of the system.

4 Concluding Remarks

We modeled a synchronous, special-purpose system as a directed graph in which the functions performed by the processors were associated to the vertices, and the computational and propagation delays to the edges. We discussed extensively the motivation for viewing the analysis problem as a conversion from this graph into a signal flow graph, and we developed a procedure that allowed us to transform this graph into a signal flow graph with the same topology as the original graph, but in which the weights in the edges correspond to the states of the system. This conversion is the reverse step to the design process. Once we have the states of the system, we can write by inspection state-like equations, and, therefore, the algorithm implemented by the system. Our procedure recovers an algorithm that is "equivalent" to the one the system was designed to implement. We formally proved our procedure and gave several examples of how to use it.

Our analysis can be extended to systems in which not all vertices are reachable from at least one source, thereby removing the constraint imposed by Remark 4 to Definition 1 (in Section 2.1). It can be shown that the logical graph of such systems always contain autonomous subgraphs that are not reachable from the rest of the graph. In order to apply our analysis procedure we need first to add edges from the root v_0 to a vertex (which is

not a sink) in every unreachable subgraph. This modification makes every vertex reachable from the root v_0 so that our analysis procedure can be applied. A detailed description and justification of this technique can be found in Jover (1985).

Finally, it should be noticed that our procedure can be significantly simplified when the system to be analyzed is a *systolic array*, i.e., a regular network of identical modules. It can be shown, in fact, that the analysis of such systems involves the application of our procedure to a *single module*. More details can be found in Jover (1985).

Acknowledgements

The authors wish to acknowledge the helpful suggestions made by Prof. Robert Schreiber and Vwani Roychowdhury. We also wish to thank Sandra M. Young for preparing the illustrations.

References

1. M.C. Chen and C.A. Mead, "Concurrent Algorithms as Space-Time Recursion Equations," *Proceedings of the USC Workshop on VLSI and Modern Signal Processing*, Los Angeles, November 1982. Also in *VLSI and Modern Signal Processing*, S.Y. Kung, H.J. Whitehouse, and T. Kailath, ed., Englewood Cliffs: Prentice Hall, 1985.

2. R. Crochiere and A.V. Oppenheim, "Analysis of Linear Digital Networks," *Proceedings of the IEEE*, vol. 63, pp. 581–595, April 1975.

3. G.B. Dantzig, "On the shortest route through a network," in *Studies in Graph Theory*, D.R. Fulkerson, ed., MAA Studies in Mathematics, Volume 11. MAA, 1975.

4. S. Even, *Graph Algorithms*, Rockville, MD.: Computer Science Press, 1979.

5. D.E. Heller and I.C.F. Ipsen, "Systolic Networks for Orthogonal Equivalence Decompositions," *SIAM Journal on Statistical and Scientific Computing*, Vol. 4, No. 2, pp. 261–269, June 1983.

6. H.V. Jagadish, R.G. Mathews, T. Kailath, and J.A. Newkirk, "A Study of Pipelining in Computing Arrays," to appear in *IEEE Trans-*

actions on *Computers*, 1986.

7. H.V. Jagadish, "Techniques for the Design of Parallel and Pipelined VLSI Systems for Numerical Computation," Ph.D. dissertation, Department of Electrical Engineering, Stanford University, December 1985.

8. J.M. Jover and T. Kailath, "A Parallel Architecture for the Kalman Filter Measurement Update," *Proceedings of IFAC 1984*, Budapest, Hungary, July 1984. An extended version is to appear in *Automatica*, 1986.

9. J.M. Jover, "On the Modeling and Analysis of Systolic and Systolic-Type Arrays," Ph.D. dissertation, Department of Electrical Engineering, Stanford University, December 1985.

10. T. Kailath, *Linear Systems*, Englewood Cliffs: Prentice Hall, 1980.

11. R.M. Karp and R.E. Miller, "Properties of a Model for Parallel Computation: Determinacy, Termination, and Queuing," *SIAM J. Appl. Math.*, Vol. 14, pp. 1390–1411, 1966.

12. D.E. Knuth, *The Art of Computer Programming*, Volume 3: Sorting and Searching. Reading, Mass.: Addison-Wesley, 1973.

13. H.T. Kung, "Let's design algorithms for VLSI systems," *Proceedings of the Caltech Conference on Very Large Scale Integration*, Randal Bryant, ed., pp. 55–90, Pasadena, January 1979.

14. H.T. Kung and C.E. Leiserson, "Systolic arrays (for VLSI)," *Sparse Matrix Proceedings 1978*, I.S. Duff and G.W. Stewart, ed., Society for Industrial and Applied Mathematics, pp. 256–282, 1979.

15. H.T. Kung, "Why systolic architectures," *Computer*, vol. 15, no. 1, pp. 37–46, January 1982.

16. H.T. Kung and W.T. Lin, "An Algebra for VLSI Algorithm Design," *Proceedings of the Conference on Elliptic Problem Solvers*, Monterey, California, January 1983.

17. S.Y. Kung, "On Supercomputing with Systolic/Wavefront Array Processors," *Proceedings of the IEEE*, Vol. 72, No. 7, pp. 867–884, July 1984.

18. C.J. Kuo, B.C. Levy, B.R. Musicus, "The Specification and Verification of Systolic Wave Algorithms," *1984 USC Workshop on VLSI Signal Processing*, Los Angeles, California, November 1984.

19. H.W. Lang, M. Schimmler, H. Schmeck, and H. Schröder, "Systolic Sorting on a Mesh-Connected Network." *IEEE Transactions on Computers*, vol. C–14, No. 7, pp. 652–658, July 1985.

20. C.E. Leiserson and J.B. Saxe, "Optimizing Synchronous Systems," *Proceedings of the XXII Symposium on the Foundations of Computer Science*, IEEE, pp. 23–36, October 1981.

21. C.E. Leiserson, F.M. Flavio, and J.B. Saxe, "Optimizing Synchronous Circuitry by Retiming," *Proceedings of the Third Caltech Conference on VLSI*, pp. 87–116, 1983.

22. H. Lev-Ari, "Modular Computing Networks: A New Methodology for Analysis and Design of Parallel Algorithms/Architectures," Integrated Systems Inc., Report #29, Palo Alto, California, December 1983.

23. C. Mead and L. Conway, *Introduction to VLSI Systems*, Reading (Mass.): Addison-Wesley, 1980.

24. S.K. Rao, "Regular Iterative Algorithms and Their Implementations on Processor Arrays," Ph.D. dissertation, Department of Electrical Engineering, Stanford University, October 1985.

25. R.A. Rohrer, *Circuit Theory: An Introduction to the State Variable Approach*, New York: McGraw-Hill, 1970.

26. R. Schreiber, "On Systolic Arrays Methods for Band Matrix Factorizations," Department of Numerical Analysis and Computing Science, The Royal Institute of Technology (Sweden), Technical Report TRITA-NA-8316, 1983.

27. E. Tidén, "Verification of Systolic Arrays—A Case Study," Department of Numerical Analysis and Computing Science, The Royal Institute of Technology (Sweden), Technical Report TRITA-NA-8403, 1984.

Appendix: EXAMPLES

In this appendix we illustrate our procedure by analyzing the arrays previously verified or analyzed in the literature. Most of the authors limit their efforts to the Kung-Leiserson matrix-multiplication array; the most comprehensive effort was made by Melhem and Rheinboldt (1984), who studied three additional examples: reverse convolution, sorting, and back substitution. We analyze these three examples below. Note that the examples studied by the previous authors always assume that all the operations take one unit of time to perform (in our language, that the computational delays are unity for all the edges of G); to make a different assumption might complicate their methods. In contrast, we can assign "realistic" values for all the computational delays, since this does not complicate our analysis procedure at all.

We also analyze the Heller-Ipsen(1983) array for QR factorization, which was studied by Tidén (1984), as an example of a complex system with regular topology.

Forward Convolution

This array was developed by Kung and Leiserson (1979). Figure 10 depicts the array, while the logical graph G is shown in Figure 11. The input

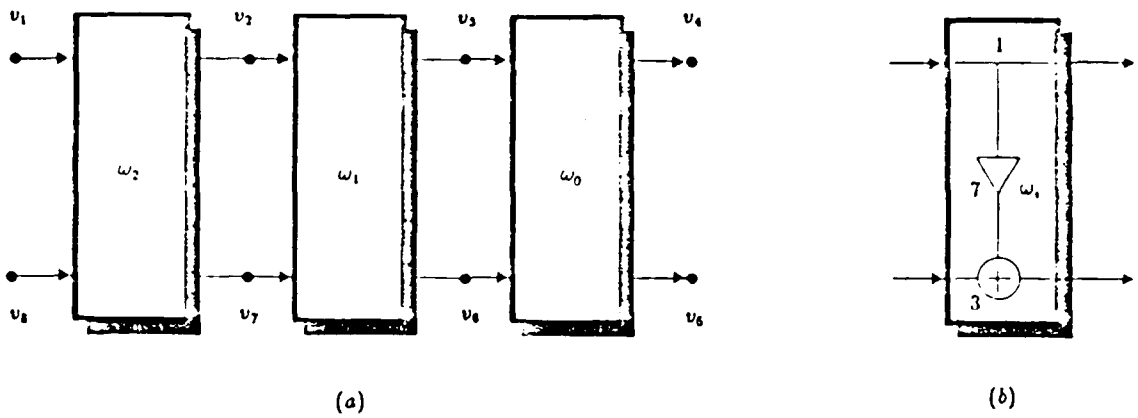


Figure 10: System that performs convolution: (a) network, (b) module and timing.

sequences are

$$u_1 = \{u_1(0), u_1(1), u_1(2), \ldots\} \tag{8}$$

$$u_8 = \{0, 0, 0, \ldots\} \tag{9}$$

We assume that $\lambda_1 = 0$ and that we know that $\lambda_8 = 7$, and the iteration interval is given as 2.

Following steps 1-2 of the analysis procedure, we determine the shortest path between v_0 and the rest of the vertices. In practice, there is no need to write v_0 since it is enough to write the λ's to keep track of the shortest path as the λ's correspond to the shortest distance from v_0 to a given vertex. Figure 12 shows one choice for the shortest-path tree. (Note that there is another choice for the shortest-path tree: take $e_{8,7}$ as part of the tree and e_{17} as a link.)

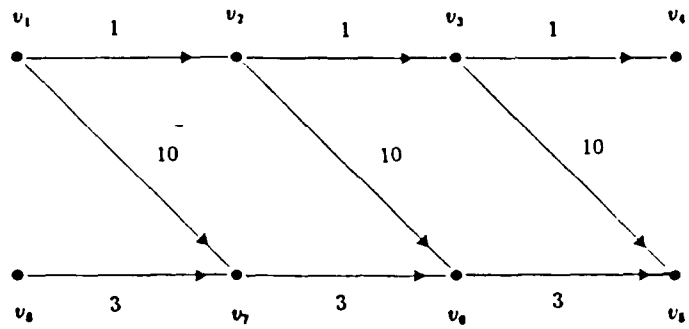


Figure 11: Graph for convolution system.

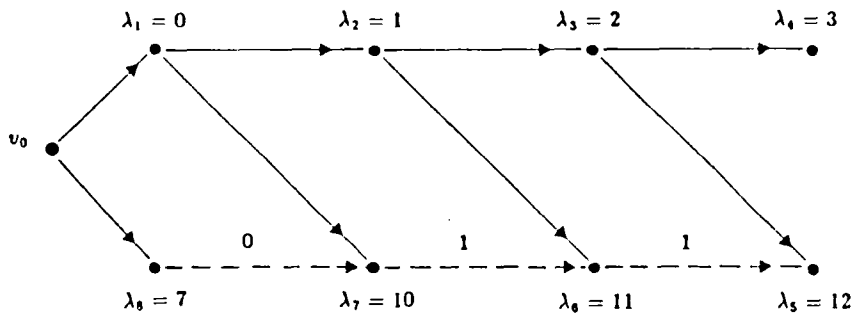


Figure 12: Shortest path tree in G_{ext} for a convolution system (links shown in dashed lines).

In steps 3-6 of the analysis procedure we compute $d^*_{76} = d^*_{65} = 2$ and $\delta_1 = gcd\{2\} = 2$; therefore there is no interleaving of sequences ($\delta_1/\delta = 1$). Furthermore, the number of states in each of the links is given by $s_{76} = s_{65} = 1$ and $s_{87} = 0$.

Finally, we can write the equations directly. Note that the equations for vertices v_5 and v_6 reflect the fact that there is a state on edges e_{76} and e_{65}:

$$\begin{cases} x_2(k) = u_1(k) \\ x_3(k) = x_2(k) \\ x_6(k) = \omega_1 x_2(k) + x_7(k-1) \\ x_7(k) = \omega_2 u_1(k) + u_8(k) \end{cases} \tag{10}$$

Substituting these equations into the following equations for the outputs

$$\begin{aligned} y_4(k) &= x_3(k) \\ y_5(k) &= \omega_0 x_3(k) + x_6(k-1) \end{aligned} \tag{11}$$

we obtain (after substituting for the input sequences, eqs. (8) and (9)

$$\begin{aligned} y_4(k) &= u_1(k) \\ y_5(k) &= \omega_0 u_1(k) + \omega_1 u_1(k-1) + \omega_2 u_1(k-2). \end{aligned} \tag{12}$$

The last equation $y_5(k) = \sum_{i=0}^{2} \omega_i u_1(k-i)$ is the output of a convolution system. Thus we have determined the algorithm performed by the network and the iteration interval for solving a single instance of the problem, $\delta_1 = 2$.

In addition, we have seen that the δ given in the statement of the problem ($\delta = 2$) was *valid* for that network; we have also seen that the network, as given, is not interleaving input data from different sequences.

Reverse Convolution

This array was developed by Kung and Leiserson (1979). We call it a "reverse" convolution array, because the output sequences are generated at opposite ends of the array. Figure 13 depicts the array, the graph G, and a shortest-path tree based on v_0. (For simplicity, we do not show the root, v_0, which has two edges: the first one connects v_0 to v_1 with delay 0; the second one connects v_0 to v_5 with delay 9.)

The input sequences are zeros at v_5 and $\{u(0), u(1), u(2), \ldots\}$ at vertex v_1, and the output sequences are $\{y(0), y(1), y(2), \ldots\}$ at v_8 and, at v_4, $\{u(-2), u(-1), u(0), u(1), u(2), \cdots\}$ with $u(-2)$ and $u(-1)$ supposed to be known (they are called *initial conditions*). The iteration interval is given as 4.

This system differs from the forward convolution in three aspects: reversal in inputs and outputs, reversal in the order of the gains ω_i, and double iteration interval. The analysis proceeds as for the forward convolution: the graph G^* has all the weights zero except for $s_{67} = s_{78} = 1$. In addition,

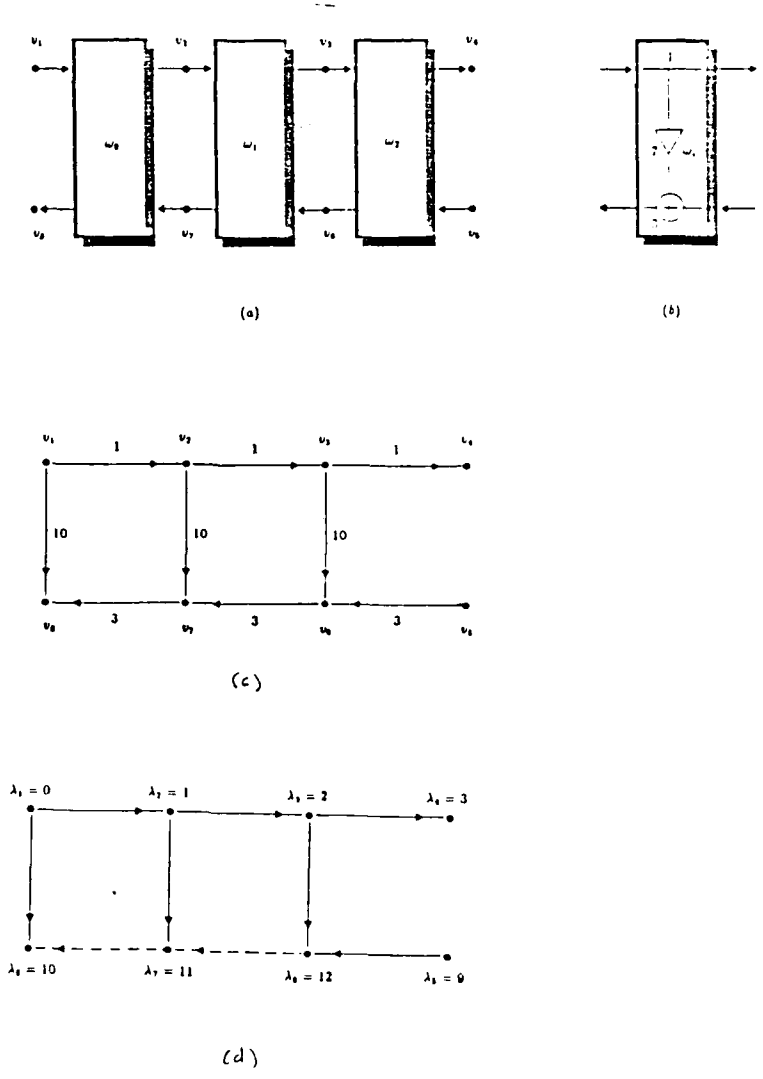


Figure 13: Reverse convolution array: (a) system, (b) processors' functions, (c) graph G, and (d) a shortest-path tree.

$\delta_1 = gcd\{4\} = 4$ so the system has a valid δ and none of the input sequences is interleaved. The system equations are given below

$$\begin{cases} x_2(k) = u_1(k) \\ x_3(k) = x_2(k) \\ x_6(k) = \omega_2 x_3(k) + u_5(k) \\ x_7(k) = \omega_1 x_2(k) + x_6(k-1) \end{cases}$$

Substituting these equations into the following equations for the outputs

$$y_4(k) = x_3(k)$$
$$y_8(k) = \omega_0 u_1(k) + x_7(k-1)$$

we obtain (after substituting for the input sequences)

$$y_4(k) = u_1(k)$$
$$y_8(k) = \omega_0 u_1(k) + \omega_1 u_1(k-1) + \omega_2 u_1(k-2)$$

The last equation $y_8(k) = \sum_{i=0}^{2} \omega_i u_1(k-i)$ is the output of a convolution system and it is the same equation that we obtained for the forward convolution.

Sorting

This array was developed by H.T. Kung and first reported and verified by Melhem and Rheinboldt (1984). The system sorts a sequence of n real num-

bers, $u_1 = \{u_1(1), u_1(2), \ldots, u_1(k)\}$ by using the linear array of $n-1$ processors depicted in Figure 14. The output sequence, $y_8 = \{y_1(1), y_1(2), \ldots, y_1(k)\}$ is sorted in ascending order. In this example, we assume all the computational delays to be equal and of value 10 since all the operations are well balanced and should take the same amount of time. The iteration interval given is 20. We have only one source, so we take it directly as the root with $\lambda_1 = 0$. Figure 14(c) and (d) show the graph and the shortest-path tree (in this case is unique). From it we can readily write the equations

$$\begin{cases} x_2(k) = max\{u_1(k), x_7(k-1)\} \\ x_3(k) = max\{x_2(k), x_6(k-1)\} \\ x_4(k) = max\{x_3(k), x_5(k-1)\} \\ x_5(k) = x_4(k) \\ x_6(k) = min\{x_3(k), x_5(k-1)\} \\ x_7(k) = min\{x_2(k), x_6(k-1)\} \\ y_8(k) = min\{u_1(k), x_7(k-1)\} \end{cases}$$

which can be rearranged as follows, after substituting $x_5(k) = x_4(k)$

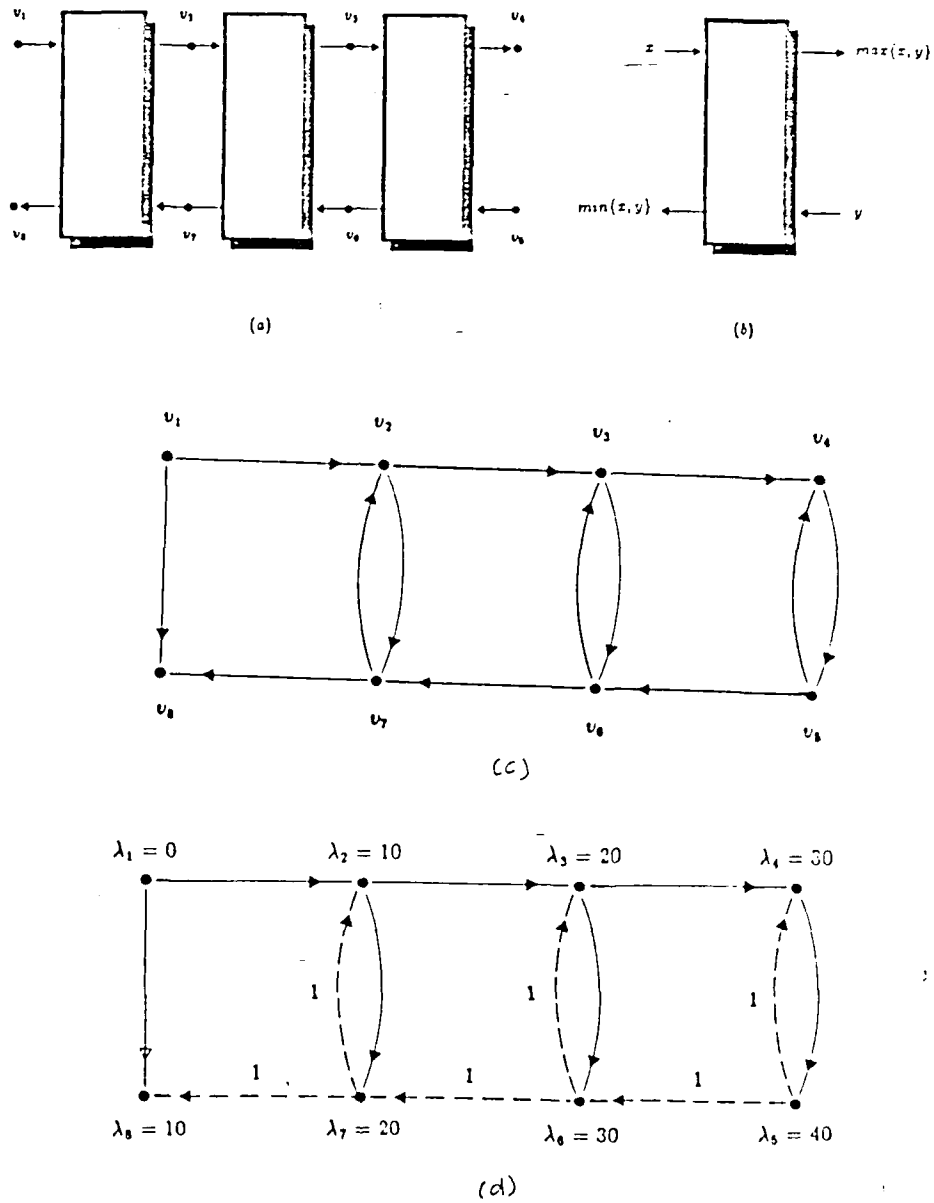


Figure 14: Sorting array: (a) system, (b) processors' functions, (c) graph G, and (d) the shortest-path tree.

$$\begin{cases} x_2(k) = max\{u_1(k), x_7(k-1)\} \\ y_8(k) = min\{u_1(k), x_7(k-1)\} \\ x_3(k) = max\{x_2(k), x_6(k-1)\} \\ x_7(k) = min\{x_2(k), x_6(k-1)\} \\ x_4(k) = max\{x_3(k), x_4(k-1)\} \\ x_6(k) = min\{x_3(k), x_4(k-1)\} \end{cases}$$

These equations correspond to the so-called bubble sort (see Knuth, 1973). Other types of sorting algorithms could be implemented, see, for instance Rao (1985) and Lang *et al.* (1985).

Back Substitution

This array was developed by Kung and Leiserson (1979) and also verified by Melhem and Rheinboldt (1984). Figure 15 depicts the system, the processors' functions, the graph G, and a shortest-path tree. This time, we have associated a computational delay of 10 for each of the edges in the graph G; this choice corresponds to the usual one for systolic arrays: all the outputs to a cell are produced at the same time even if some of the outputs may take less time to compute.

The input sequences are as follows: zeros at vertex v_6, the elements

of a column vector, b, at vertex v_1, and the diagonals of a banded, lower-triangular matrix, A at vertices v_2–v_5 (the main diagonal at v_2, the first subdiagonal at v_3, and so on). The input sequences are as follows:

$$\begin{cases} u_1 = \{b_1, b_2, b_3, \ldots\} \\ u_2 = \{a_{11}, a_{22}, a_{33}, \ldots\} \\ u_3 = \{a_{21}, a_{32}, a_{43}, \ldots\} \\ u_4 = \{a_{31}, a_{42}, a_{53}, \ldots\} \\ u_5 = \{a_{41}, a_{52}, a_{63}, \ldots\} \end{cases}$$

We are also given the times, λ, at which the first input is entered; they are as follows

$$\lambda_1 = \lambda_2 = 30, \quad \lambda_3 = 40, \quad \lambda_4 = 50, \quad \lambda_5 = 60, \quad \lambda_6 = 0.$$

These times correspond to the weights in the edges connecting the root (not shown) and the input vertices.

The output is at vertex v_{13}; we can write it as

$$y_{13} = \{y(1), y(2), y(3), \ldots\}$$

Figure 15(d) shows the states in addition to the shortest path. We computed them using $\delta_1 = gcd\{20, 40, 60\} = 20$. From this figure we can write the following equations

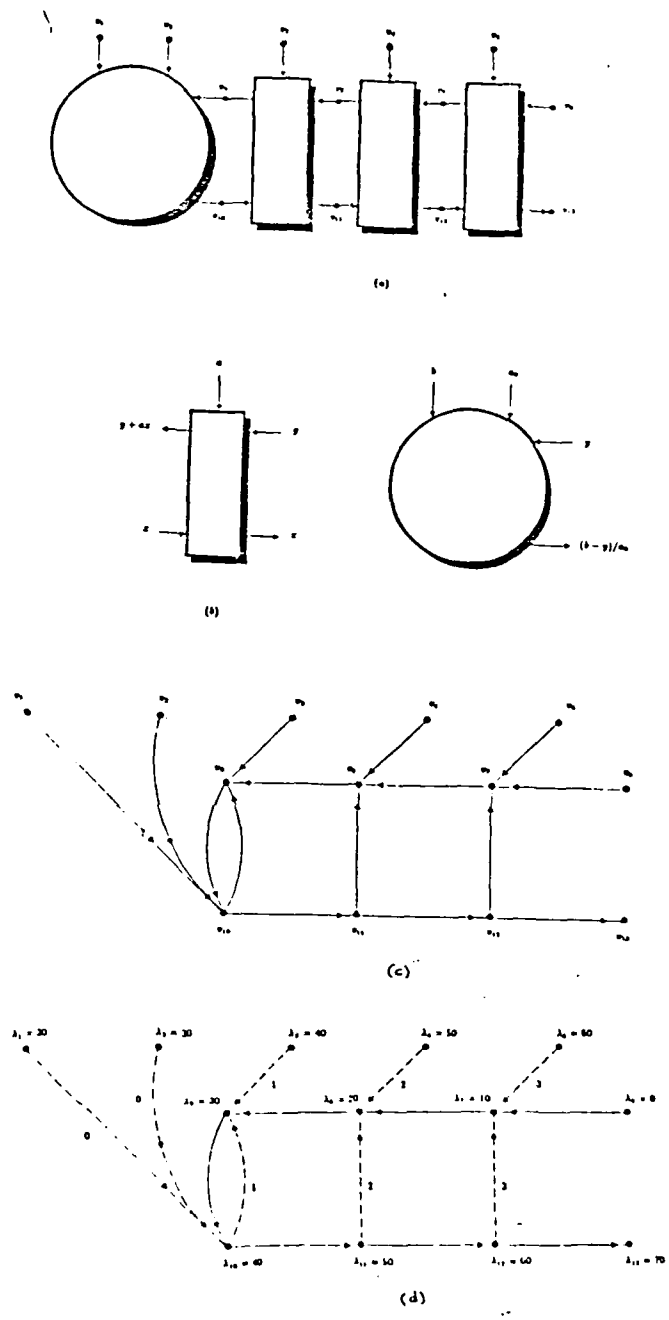


Figure 15: Back substitution array: (a) system, (b) processors' functions, (c) graph G, and (d) a shortest-path tree.

$$\begin{cases} x_7(k) &= x_{12}(k-3) \cdot u_5(k-3) + u_6(k) \\ x_8(k) &= x_{11}(k-2) \cdot u_4(k-2) + x_7(k) \\ x_9(k) &= x_{10}(k-1) \cdot u_3(k-1) + x_8(k) \end{cases}$$

$$\begin{cases} x_{10}(k) &= (u_1(k) - x_9(k))/u_2(k) \\ x_{11}(k) &= x_{10}(k) \\ x_{12}(k) &= x_{11}(k) \\ y_{13}(k) &= x_{12}(k) \end{cases}$$

Substituting these equations we get

$$y_{13}(k) = (u_1(k) - x_9(k))/u_2(k)$$

where

$$x_9(k) = y(k-1) \cdot u_3(k-1) + y(k-2) \cdot u_4(k-2) + y(k-3) \cdot u_5(k-3) + u_6(k)$$

Substituting now into the input and output sequences we obtain

$$y_{13}(k) = (b_k - x_9(k))/a_{kk}$$

where

$$x_9(k) = y(k-1) \cdot a_{k,k-1} + y(k-2) \cdot a_{k,k-2} + y(k-3) \cdot a_{k,k-3}$$

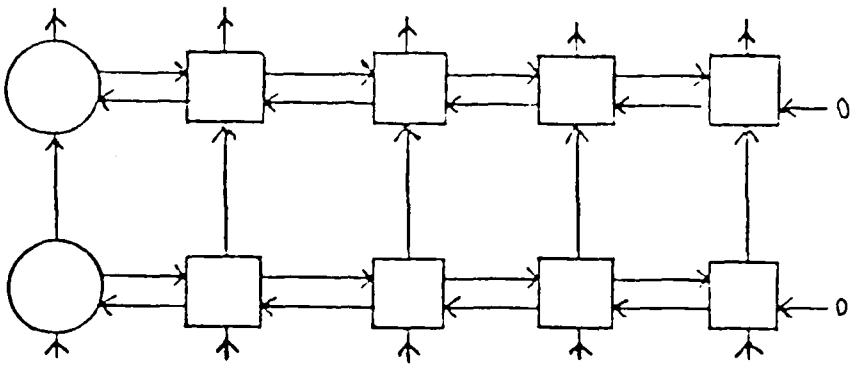
These equations are the well known method of solving a triangular system, called back substitution. Therefore, the array discussed performs back substitution.

QR factorization

Our last example is the Heller-Ipsen (1983) array for QR factorization of a banded matrix. A more general study of such arrays can be found in Schreiber (1983). The Heller-Ipsen array has been verified by Tidén (1984) using a novel approach: he shows that the system works with one module ('size' of the system = 1); then he applies induction to the size of the array, assuming that it works for size n, and proves that it works for size $n+1$.

Figure 16 depicts the array and the inputs for the matrix A; the numbers indicate the subindex for the elements of this matrix. Figure 17 shows the naming convention and the coordinate axis for the variables, and gives the graph G^* for a row of processors. Note the variations of the graph at the boundaries and in some of the weights in the subdiagonal cell.

All the computational delays are one unit of time. The iteration interval is given as 2, which is valid because $\delta_1 = gcd\{0,2,4\} = 2$. With the information in Figure 17 we can write directly the equations for the system. Note that the first two coordinates indicate position and the third one time.



0	#	11	#	0
#	21	#	12	#
31	#	22	#	13
#	32	#	23	#
42	#	33	#	24
#	43	#	34	#
53	#	44	#	35
#	54	#	45	#
0	#	55	#	0
#	0	#	0	#

Figure 16: QR factorization array (as given in Tidén, 1984).

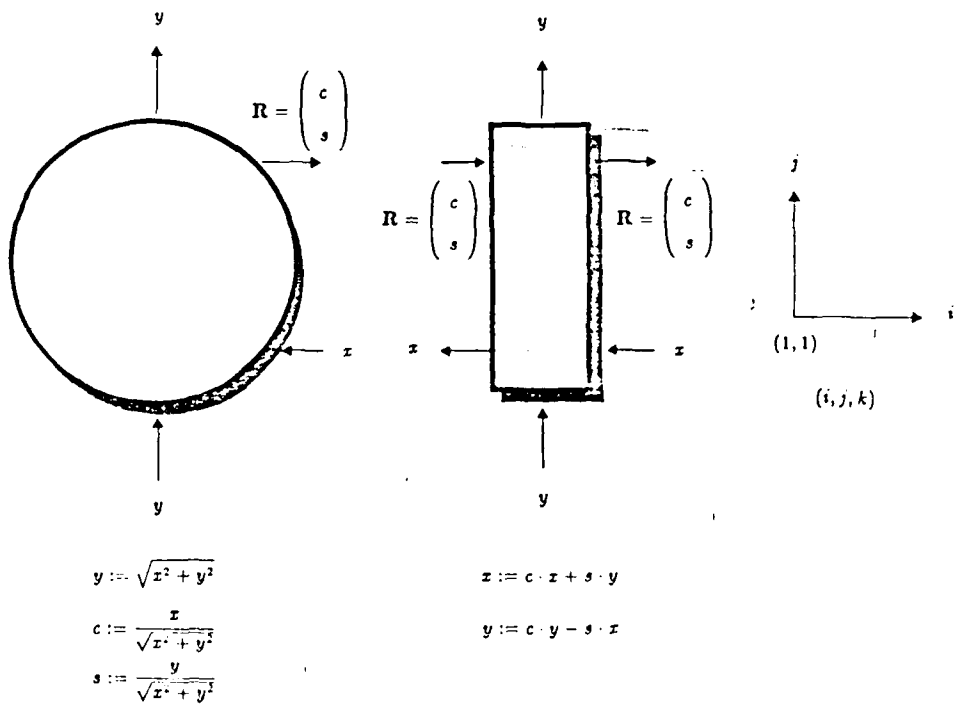


Figure 17: QR factorization details: (a) naming convention and coordinate axis, and (b) graph G^* for a row of processors.

We write below the regular iterative algorithm for a system with an input matrix A that has N subdiagonals, a main diagonal, and N superdiagonals.

$$\text{Initialization:} \qquad y(i,1,k) = \begin{cases} a_{k+i-N-1,k}, & i \le N+1 \\ a_{k,k+i-N-1}, & i > N+1 \end{cases}$$

For $\quad j = 1, 2, \ldots N$

$$\mathbf{R}(2,j,k) = (x^2(1,j,k) + y^2(1,j,k))^{-1/2} \begin{pmatrix} x(1,j,k) \\ y(1,j,k) \end{pmatrix}$$

$$y(1,j+1,k) = (x^2(1,j,k) + y^2(1,j,k))^{1/2}$$

For $\quad i = 2, 3, \ldots 2N+1$

$$\mathbf{R}(i+1,j,k) = \mathbf{R}(i,j,k)$$

$$x(2N+1,j,k) = 0$$

$$x(i-1,j,k) = \begin{cases} (x(i,j,k-1) \quad y(i,j,k))\mathbf{R}(i,j,k-1), & i = 2 \\ (x(i,j,k-1) \quad y(i,j,k))\mathbf{R}(i,j,k-2), & i > 2 \end{cases}$$

$$y(i,j+1,k) = \begin{cases} (y(i,j,k) \quad -x(i,j,k))\mathbf{R}(i,j,k-1), & i = 2 \\ (y(i,j,k) \quad -x(i,j,k-1))\mathbf{R}(i,j,k-2), & i > 2 \end{cases}$$

These equations correspond to the QR factorization of a banded matrix, which completes our example.

END

1-87

DTIC